SDEV1201 Database Fundamentals

LESSON 16

Let’s review

Anonymous Code Blocks

Why use DO?

Run multi-statement logic without creating a stored function/procedure.

Perfect for data fixes, quick loops, conditional DDL, and ad-hoc scripting.

Run multi-statement logic without creating a stored function/procedure.

Anonymous Code Blocks

```
--$$ create the block for the code that you will execute
DO $$
Declare --A section to declare variables
    my_message TEXT:= 'Hello World!'; --Declares a variable called my_message and assigns 'Hello World to it'
    my_mood TEXT:= 'groovy';
Begin--The executable statements go here
Raise Notice '% Hope you have a % day!', my_message,my_mood;

--Raises a message to pgamdins messages tab or whatever output console you are working in. It is mainly for testing and debugging.
--% is a place holder
--You can also format a raise Notice as
Raise Notice using message = my_message ||' Hope you have a ' || my_mood || 'day!';

End $$;
```

Variables

Variables

Variables are used to hold and manipulate values in memory. They must be declared before they are used, allowing PostgreSQL to allocate storage efficiently.

Syntax

```
Declare
   variablename datatype;
   variablename datatype;
   variablename datatype;
```

All variables must be declared at the beginning of the block, before any executable statements.

Variables

Assigning Values to Variables

Variables can be assigned literal values or values from queries.

Assigning a Literal Value

variable_name := literal_value;

Flow Control

You can control the execution of SQL statements using flow control structures such as If, ElsIf (optional), and Else (optional). You can have as many ElsIf statements as you want, but only one Else statement is allowed. You can use Begin and End for readability, but it is not required in If statements.

Flow Control

Syntax:

If **condition** Then
  Begin
    -- SQL statements
  End
ElsIf **another_condition** Then
  Begin
    -- SQL statements
  End
Else
  Begin
    -- SQL statements
  End
End If;

Flow Control

Anonymous Code Blocks (Do)

Each Do block or stored procedure executes as a single unit.

You can separate logical sections of a script simply by executing one block at a time.

Hypothetical Example (Client table does not exist in IQSchool):

```
Do $$
Begin
   Alter Table Client
      Add Column Active Char(1);
End $$;

Do $$
Begin
   Alter Table Client
      Add Constraint CK_Client_Active_TF Check (Active In ('T', 'F'));
End $$;
```

In this example, each Do block executes separately. This ensures the column is added before the constraint is created.

Example

1. Create an anonymous code block. Declare a variable with a clubID in it. If the count of students in that club is greater than 2 Raise Notice ‘A successful club!’. If the count is not greater than 2 Raise Notice ‘Needs more members!’.

```
Do $$
Declare
    club_id varchar(10) := 'ACM';
    member_count int;
Begin
    Select count(*)
    into member_count
    from activity
    where LOWER(clubid) = LOWER(club_id);----could use to avoid case errors
    if member_count > 2 Then
    Raise Notice 'A succesful club!';
    else
    Raise Notice 'Needs more members!';
    End If;
End $$;

--lets see how many are in each club first to test
Select clubid, count(*)
from activity
group by clubid;
```

Example

2. Create an anonymous code block. Create 2 variables to hold a students first and last name. If the student's name in the variables is already in the table, then Raise Notice 'We already have a student with the name firstname lastname!' Otherwise Raise Notice 'Welcome firstname lastname!'

```
Do $$
Declare
  First_Name varchar(30) := 'Greg';
  Last_Name varchar(30) := 'Henke';
Begin
  If Exists (select 1
          from student
          where FirstName = First_Name
          and LastName = Last_Name) then
    Raise Notice 'We already have a student with the name % %!', First_Name, Last_Name;
  else
    Raise Notice 'Welcome % %!', First_Name, Last_Name;
End If;
End $$;

-- Check enrolled students
select FirstName, LastName from Student
```

Activity

Work on “SQL Flow Control and Variable Exercises (PostgreSQL)” which is located in Brightspace “Week 10 & 11 – Stored Procedures” under Activities.

Question 1

Create a "Do" block to display the current date and time.

```
Do $$
Declare
    v_Today date := Current_Date;
    v_Now   time := Current_Time;
Begin
    Raise Notice 'Today''s date is %', v_Today;
    Raise Notice 'Current time is %', v_Now;
End;
$$;
```

Question 2

Create a "Do" block to display how many students exist in the Student table.

```
Do $$
Declare
    v_Total integer;
Begin
    Select    Count(StudentID)
      Into    v_Total
    From      Student;

    Raise Notice 'Total number of students: %', v_Total;
End;
$$;
```

Question 3

Create a "Do" block to display a message if there are no records in the Payment table.

```
Do $$
Declare
   v_Count integer;
Begin
   Select   Count(*)
     Into     v_Count
   From     Payment;

   If v_Count = 0 Then
      Raise Notice 'No payment records found.';
   Else
      Raise Notice '% payment record(s) found.', v_Count;
   End If;
End;
$$;
```

Question 4

Create a "Do" block to add up and display all payment amounts in the Payment table and display the total collected.

```
Do $$
Declare
   v_Total decimal(10,2);
Begin
   Select   Sum(Amount)
    Into v_Total
   From     Payment;

   If v_Total Is Null Then
      v_Total := 0;
   End If;

   Raise Notice 'Total payments collected is: $%', v_Total;
End;
$$;
```

Question 5

Create a "Do" block to display if there are any courses with no students registered, or if all courses have registrations.

```
Do $$
Declare
    v_Missing integer;
Begin
    Select Count(*)
    Into v_Missing
    From Course
    Where CourseID Not In (Select CourseID
     From Registration);
    If v_Missing > 0 Then
        Raise Notice 'There are % course(s) with no registrations.', v_Missing;
    Else
        Raise Notice 'All courses have at least one registration.';
    End If;
End;
$$;
```

Today’s Objective

By the end of this section you will be able to:

- Write **stored procedures**

Documentation can be found in Brightspace in the “Week 10 & 11 – Stored Procedures” section under Materials. “Stored Procedures and Functions – Simple (PostgreSQL Version)”

Assignment 3 (TH : Create Table & DML Statements) is available on Brightspace and is **due Friday, October 31, 2025 at 5:00 PM.**

Stored Procedures

Stored Procedures

A stored procedure is a set of SQL statements. It is very similar to other procedures that you may have used in other languages. The stored procedure has a name and can accept parameters.

A stored procedure in PostgreSQL is a named block of SQL statements that can be executed as a single unit. Stored procedures allow you to group related logic together, accept parameters, and control program flow; much like procedures in other programming languages.

They're compiled into machine language and stored in the database.

After it's been created, a SP can be executed without having to be re-compiled each time as a script would.

Stored Procedures

Characteristics of Stored Procedures

- Stored procedures are **saved inside the database** and can be called repeatedly without re-parsing or re-planning each time (which improves performance).
- The **source code** of the procedure is stored in the database system catalogs.
- Once created, a stored procedure can be executed without recompiling the SQL each time.
- **Important**: Even though the source code is stored in the database, it is good practice to keep an external copy of your procedure code for backup or version control.

Stored Procedures

Simplified Syntax

In PostgreSQL, stored procedures are created using the CREATE PROCEDURE statement. They are written using the PL/pgSQL procedural language.

Syntax:

```
CREATE PROCEDURE procedure_name ( [parameter_list] )
LANGUAGE plpgSQL
AS $$body$$
BEGIN
    -- SQL statements
END;
$$body$$;
```

Stored Procedures

Notes:

- The LANGUAGE plpgSQL clause specifies that the procedure uses PostgreSQL’s procedural language.
- $$body$${...}$$body$$ marks the code block. With $$, you don’t need to escape single quotes inside the block. The code looks cleaner and easier to read.
- Unlike SQL Server, PostgreSQL does not use RETURN in procedures. Instead, to return values, create a function (using CREATE FUNCTION).

Stored Procedures

Example: Simple Stored Procedure

```
Create Procedure HelloWorld()
Language plpgSQL
As $body$
Begin
   Raise Notice '%', 'Hello, World!';
End;
$body$;
```

Executing the Procedure

To run a procedure, use the CALL command:

```
Call HelloWorld();
```

Stored Procedures

Stored Procedure Capabilities

A PostgreSQL stored procedure can include:

- **DML statements** (INSERT, UPDATE, DELETE, SELECT)
- **Flow control** (IF, ELSIF, LOOP, etc.)
- **Transactions** (procedures can explicitly COMMIT or ROLLBACK)
- **Exception handling** (BEGIN … EXCEPTION … END)
- **Variable declarations and assignments**

Stored Procedures

Modifying a Stored Procedure

You can update an existing procedure in one of two ways:

1. **Drop and Recreate**

Dropping removes the procedure completely from the database.

DROP PROCEDURE procedure_name ([parameter_types]);

Then recreate it using CREATE PROCEDURE.

Stored Procedures

2. Use ALTER PROCEDURE

You can modify some attributes (such as the owner or schema) or replace the procedure definition using CREATE OR REPLACE PROCEDURE.

CREATE OR REPLACE PROCEDURE **procedure_name** ([parameter_list])
LANGUAGE plpgSQL
AS $body$
BEGIN
  -- Updated SQL statements
END;
$body$;

Stored Procedures

Note: CREATE OR REPLACE PROCEDURE replaces the procedure body (between Begin and End) but does not overwrite anything else. If the input arguments are different, PostgreSQL creates a new procedure with the same name and different input arguments.

Example

```
Create Procedure PR_UpdateWithdrawStatus()
Language plpgSQL
As $body$
Begin
   Update Registration
   Set WithdrawYN = 'Y'
   Where Mark < 50;
End;
$body$;
```

Example

Invoking Stored Procedures

To execute a procedure:

CALL procedure_name ([arguments]);

Example:

Call PR_UpdateWithdrawStatus();

Functions

If you need to return a value, you will want to use a function instead. Stored functions are created using the CREATE FUNCTION statement. They are also written using the PL/pgSQL procedural language.

Functions

Simplified Syntax:

```
CREATE FUNCTION function_name ( [parameter_list] )
Returns data_type
LANGUAGE plpgSQL
AS $$
BEGIN
   -- SQL statements
   Return value;
END;
$$;
```

Functions

Example:

```
Create Function FN_GetTotalPayments()
Returns decimal(8,2)
Language plpgSQL
As $body$
Declare
   v_total decimal(8,2);
Begin
   Select Sum(Amount)
   Into v_total
   From Payments;
   Return v_total;
End;
$body$;
```

Functions

Invoking Stored Function

To run a function:

```
Select FN_GetTotalPayments();
```

Activity

Work on “Stored Procedures and Functions No Parameter Exercises (PostgreSQL)” which is located in Brightspace “Week 10 & 11 – Stored Procedures” under Activities.

Reminder

Assignment 3 (TH : Create Table & DML Statements) is available on Brightspace and is **due Friday, October 31, 2025 at 5:00 PM.**

Question 1

If a club named 'Default Club' does not exist, create a procedure to insert the record with a ClubID 'DEF'. If the record already exists, display the message 'A Default Club already exists’.

```
Create Procedure PR_AddDefaultClub()
Language plpgsql
As $body$
Begin
    If Not Exists (Select 1
      From Club
      Where ClubName = 'Default Club') Then
        Insert Into Club (ClubId, ClubName)
        Values ('DEF', 'Default Club');
    Else
        Raise Notice 'A Default Club already exists';
    End If;
End;
$body$;

Call PR_AddDefaultClub();
```

Question 2

Create a procedure to reduce every student’s BalanceOwing by 10% if they owe more than $0.If there are no records to reduce, display the message 'There are no records to reduce’.

```
Create Procedure PR_DiscountAllBalances()
Language plpgsql
As $body$
Begin
    If Exists (Select 1
     From Student
     Where BalanceOwing > 0) Then
       Update Student
       Set BalanceOwing = BalanceOwing * 0.90
       Where BalanceOwing > 0;
    Else
       Raise Notice 'There are no records to reduce’;
    End If;
End;
$body$;

Call PR_DiscountAllBalances();
```

Question 3

Create a procedure to delete all rows from Registration where WithdrawYN = 'Y'. If there are no records to delete, display the message 'There are no records to delete’.

```
Create Procedure PR_DeleteWithdrawnRegistrations()
Language plpgsql
As $body$
Begin
    If Exists (Select 1
     From Registration
     Where WithdrawYN = 'Y') Then
       Delete From Registration
       Where WithdrawYN = 'Y’;
    Else
       Raise Notice 'There are no records to delete’;
    End If;
End;
$body$;

Call PR_DeleteWithdrawnRegistrations();
```

Question 4

Create a function to return the total number of courses available in the Course table.

```
Create Function FN_TotalCourses()
Returns integer
Language plpgsql
As $body$
Declare
    v_Total integer;
Begin
   Select Count(CourseID)
   Into v_Total
   From Course;

    Return v_Total;
End;
$body$;

select FN_TotalCourses();
```

Question 5

Create a function to return the highest course cost charged for any course.

```
Create Function FN_HighestCourseCost()
Returns decimal(10,2)
Language plpgsql
As $body$
Declare
    v_Max decimal(10,2);
Begin
   Select Max(CourseCost)
   Into v_Max
   From Course;

   Return v_Max;
End;
$body$;

Select FN_HighestCourseCost();
```

Question 6

Create a function to return a string showing the semester name based on the current month. If the month number is between 1 and 4, the semester name is Winter; if the month number is between 5 and 8, the semester name is Summer; else the semester name is Fall.

```
Create Function FN_CurrentSemester()
Returns varchar
Language plpgsql
As $body$
Declare
   v_Month integer;
   v_Label varchar(6);
Begin
   v_Month := date_part('month', current_date);

   If v_Month Between 1 And 4 Then
      v_Label := 'Winter';
   Elsif v_Month Between 5 And 8 Then
      v_Label := 'Summer';
   Else
       v_Label := 'Fall';
   End If;

    Return v_Label;
End;
$body$;

select FN_CurrentSemester();
```

Homework

Assignment 3 (TH : Create Table & DML Statements) is available on Brightspace and is **due Friday, October 31, 2025 at 5:00 PM.**